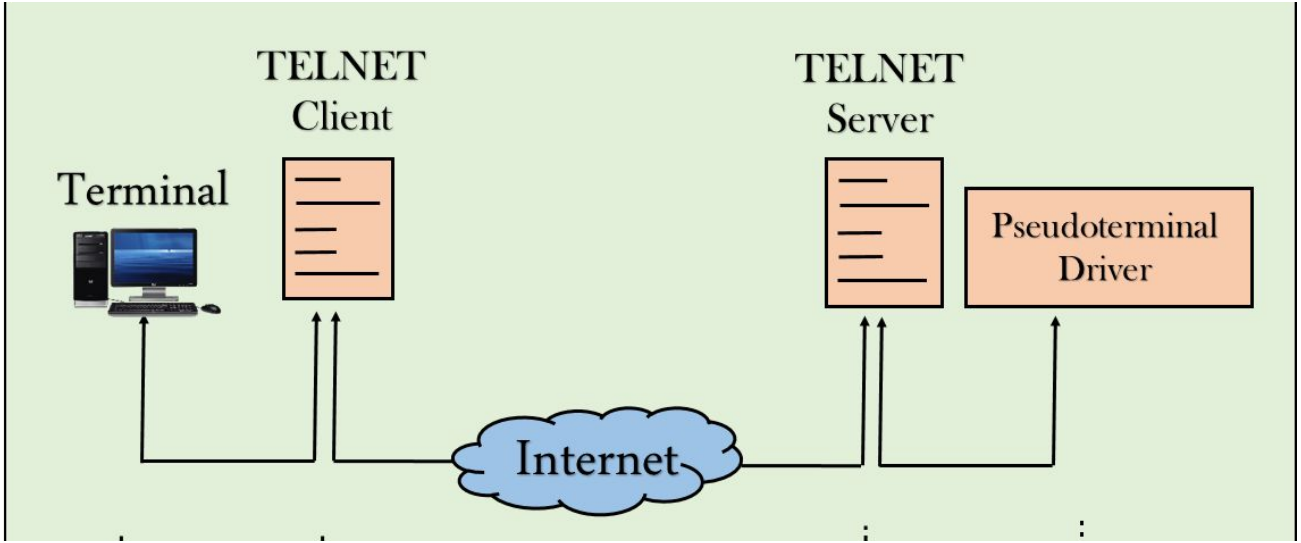
CS111

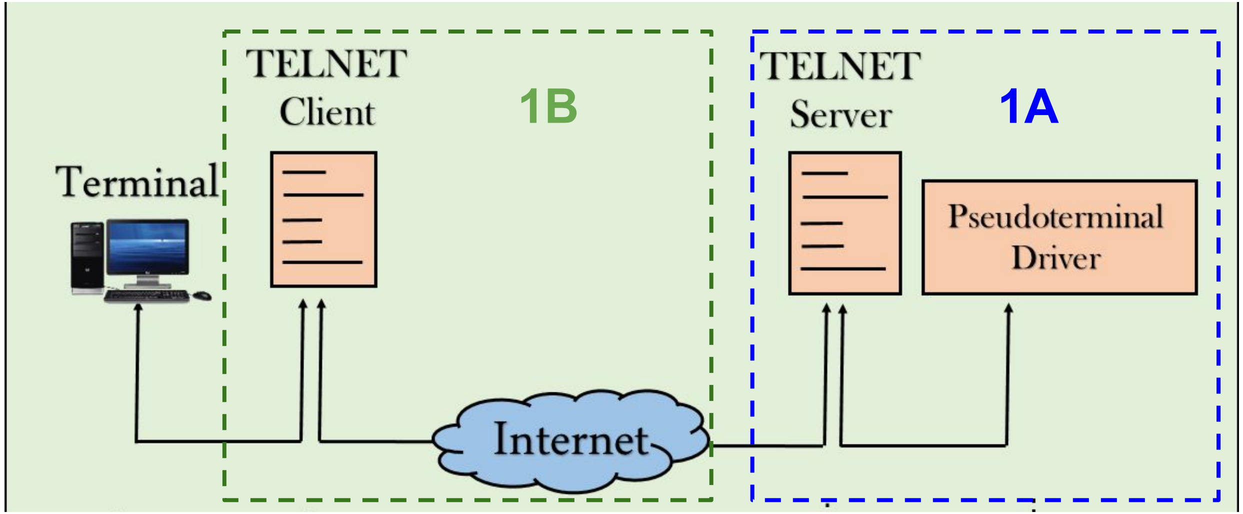
Week 3

Rishab Ketan Doshi

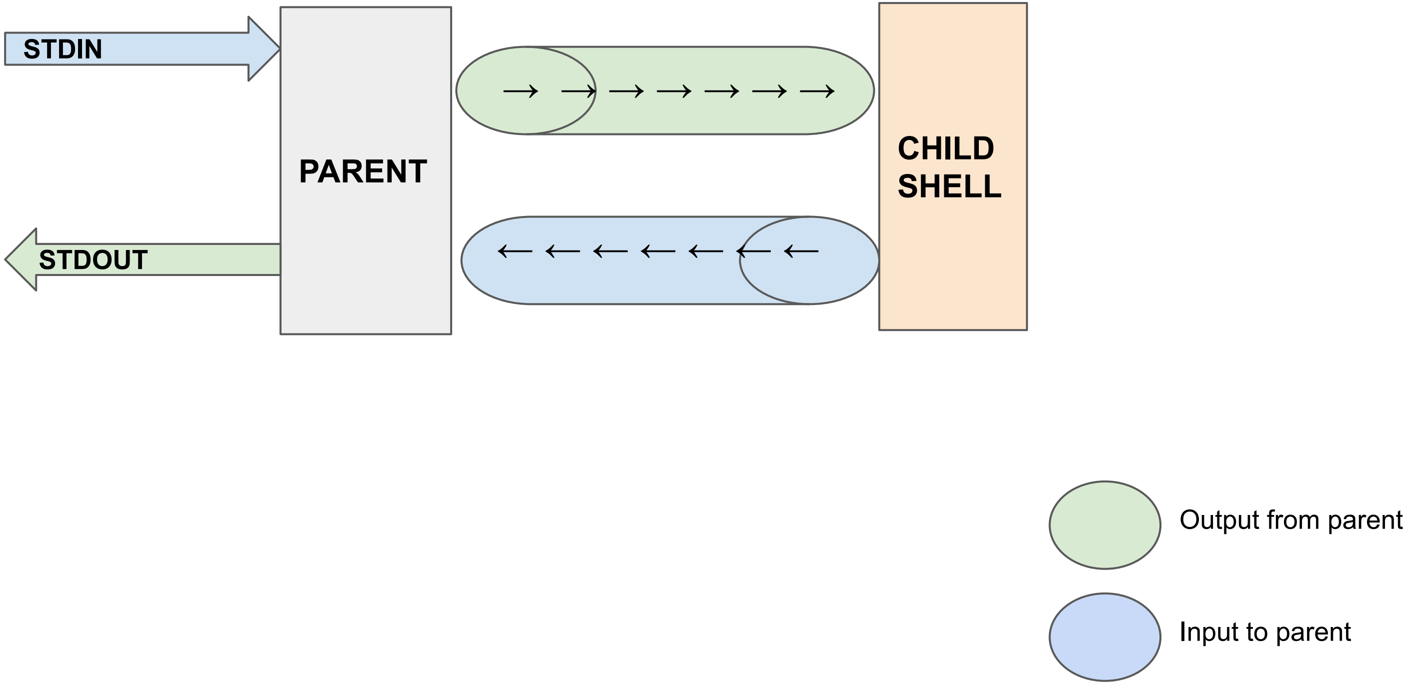
Project 1 - Telnet



Project 1A - Where are we?



Project 1A - Recap



Project 1A - parent

Change keyboard input to character at a time, i.e., non-canonical input mode

- termios, tcssetattr, tcsgetattr

Create a child shell process and execute commands

- fork() to create, exec() to execute

Read input from keyboard pass it to child shell after processing

Read processed output from child shell

Project 1A - parent

Read input from keyboard

1. Display input on stdout (screen)
2. Send input to child-shell

Display processed output from child shell

3. Output of child-shell on processing must be shown on stdout (screen)

Ex:

Project 1A - parent

Special cases:

If Ctrl+D is received from keyboard:

- Stop reading input from keyboard
- Process any remaining output from child shell
- Restore normal terminal modes
- Report the status
- Exit the program

Project 1A - parent

Special cases:

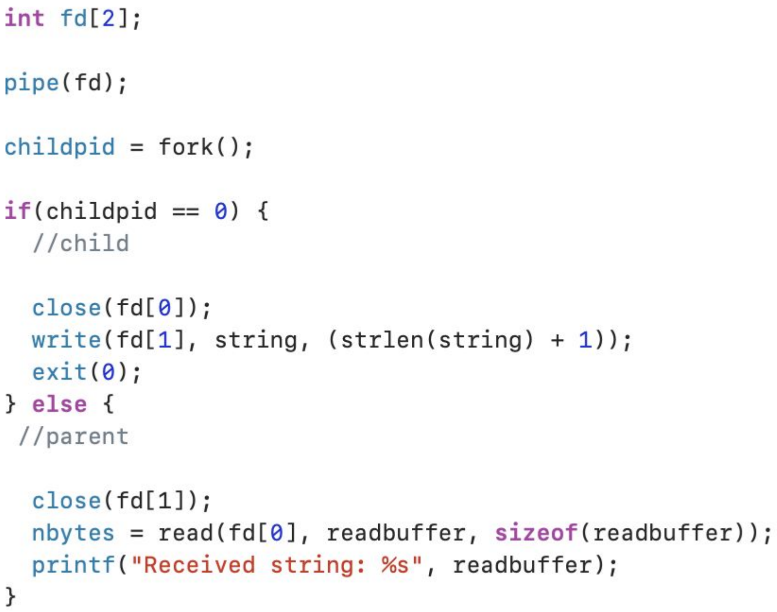
If Ctrl+C is received from keyboard:

- Stop reading input from keyboard
- Use kill(2) to send a SIGINT to shell program
- Process any remaining output from child shell
- Restore normal terminal modes
- Report the status
- Exit the program

More special cases - <cr> / <lf> to <cr><lf>

Project 1A - communication - Pipes recap

- Unidirectional flow of data from one process to another.
- Buffers data till read end of pipe doesn’t consume data present at write end of the pipe.



```
int fd[2];

pipe(fd);

childpid = fork();

if(childpid == 0) {
  //child

  close(fd[0]);
  write(fd[1], string, (strlen(string) + 1));
  exit(0);
} else {
 //parent

  close(fd[1]);
  nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
  printf("Received string: %s", readbuffer);
}
```

Pipes

Writing from child to parent:

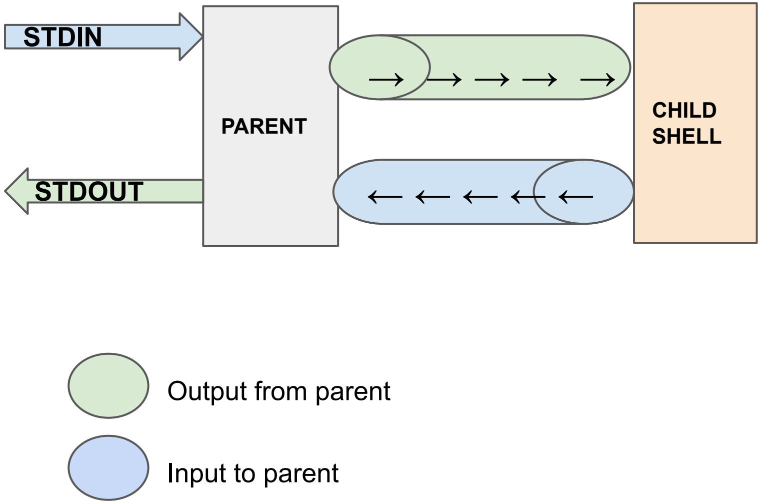
Child: I don’t need read end of pipe, I will close it

Parent: I don’t need write end of pipe, I will close it

Child: I want to send a message to parent, I will write to write end of pipe

Parent: I want to receive messages from child, I will read from read end of pipe

Project 1a - pipes recap



2 pipes:

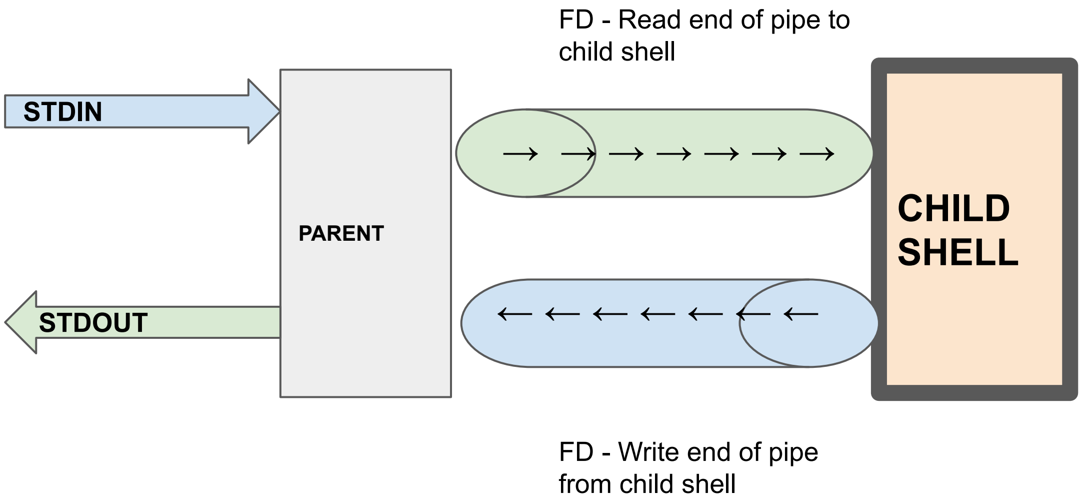
Write to child shell 🟩

Read from child shell 🟦

Parent, Child:

- I don’t need to read from which pipe?, I can close the read end of that pipe.
- I don’t need to send data to which pipe?, I can close the write end of that pipe.

Project 1a - File Descriptors - Child



FD - Read end of pipe to child shell

FD - Write end of pipe from child shell

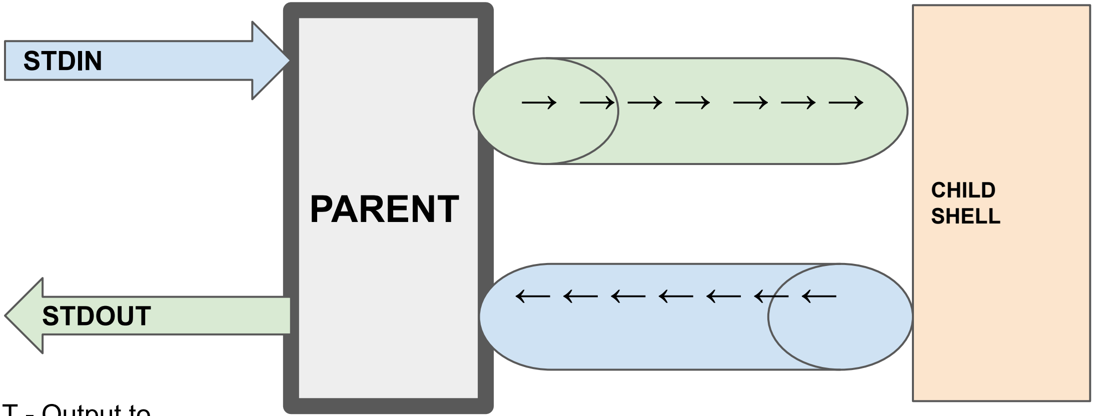
Redirect / Change stdin and stdout of child shell to the appropriate pipe ends.

Recall dup()

Project 1a - File Descriptors - Parent

STDIN - Input from
Keyboard

FD1 - Write end of pipe to
child shell



STDIN

PARENT

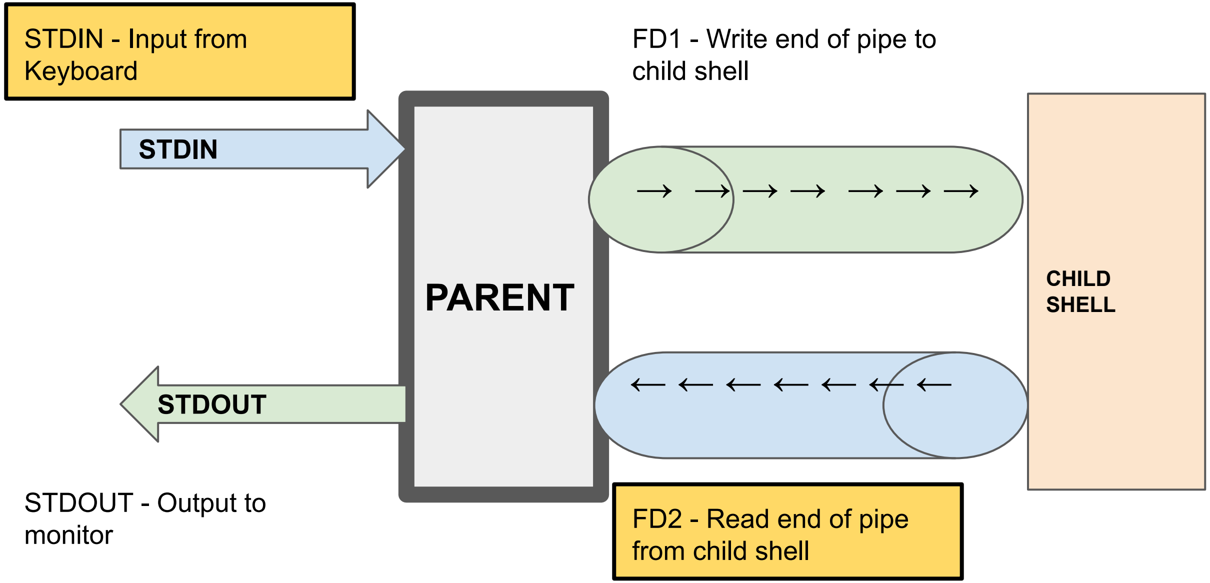
CHILD
SHELL

STDOUT

STDOUT - Output to
monitor

FD2 - Read end of pipe
from child shell

Project 1a - Poll



STDIN - Input from Keyboard

STDIN

FD1 - Write end of pipe to child shell

PARENT

CHILD SHELL

STDOUT

STDOUT - Output to monitor

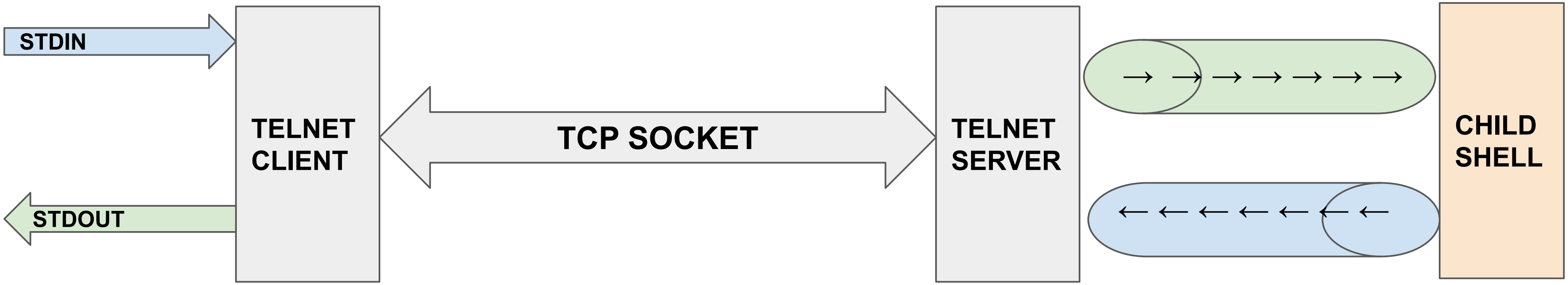
FD2 - Read end of pipe from child shell

Parent reading from 2 inputs (highlighted in yellow).

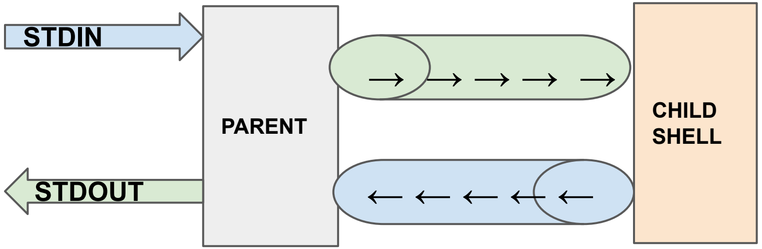
Need to poll between the file descriptors to see if there is any new input

poll()

Project 1b

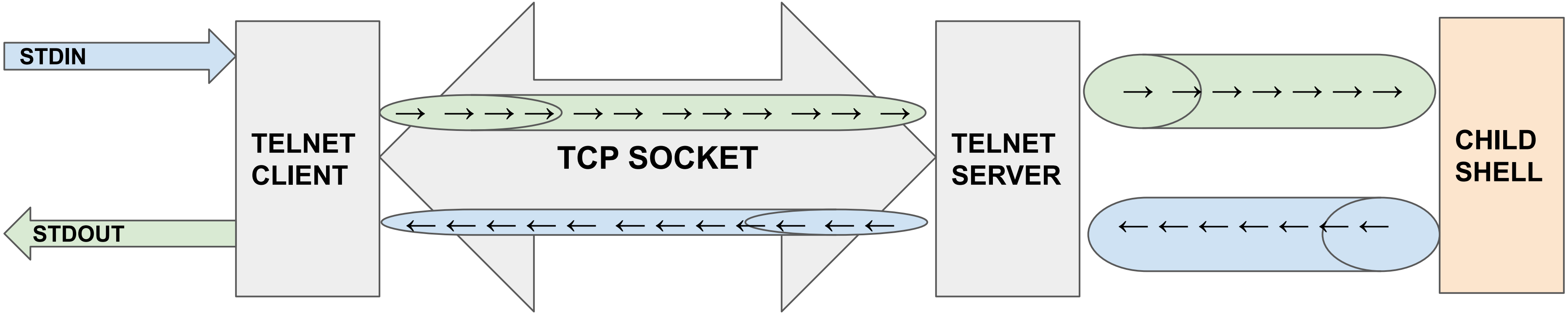

STDIN
STDOUT
TELNET CLIENT
TCP SOCKET
TELNET SERVER
CHILD SHELL

Project 1b

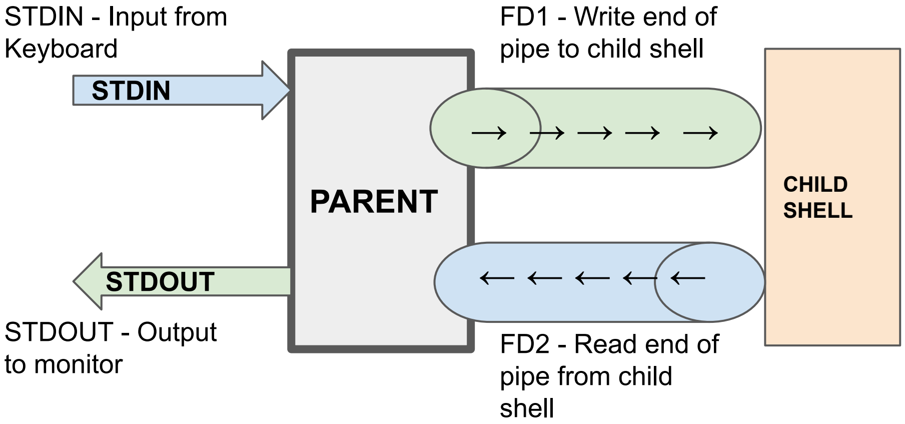


TCP Socket -

- Socket is again a file descriptor
- Full duplex - you can read and write from the same file descriptor



Project 1b - File descriptors (TELNET client)



STDIN - Input from Keyboard

STDIN

PARENT

FD1 - Write end of pipe to child shell

CHILD SHELL

STDOUT

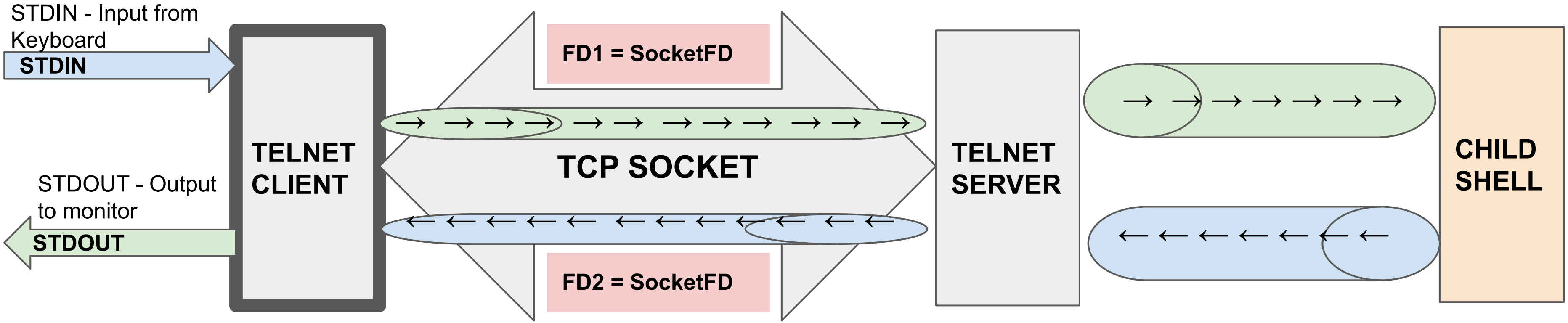
STDOUT - Output to monitor

FD2 - Read end of pipe from child shell

TCP Socket -

- Socket is again a file descriptor
- Full duplex - you can read and write from the same file descriptor

🟥 Modified file descriptors



STDIN - Input from Keyboard

STDIN

TELNET CLIENT

STDOUT - Output to monitor

STDOUT

FD1 = SocketFD

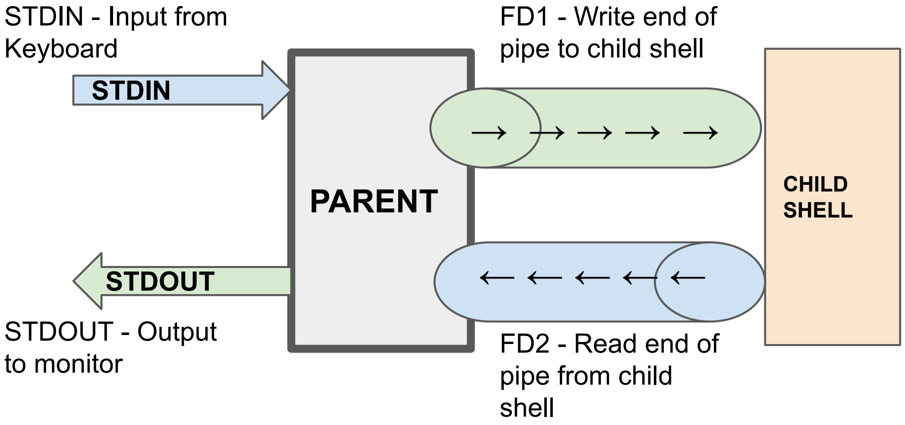
TCP SOCKET

FD2 = SocketFD

TELNET SERVER

CHILD SHELL

Project 1b - File descriptors (TELNET server)



STDIN - Input from Keyboard

STDIN

PARENT

STDOUT

STDOUT - Output to monitor

FD1 - Write end of pipe to child shell

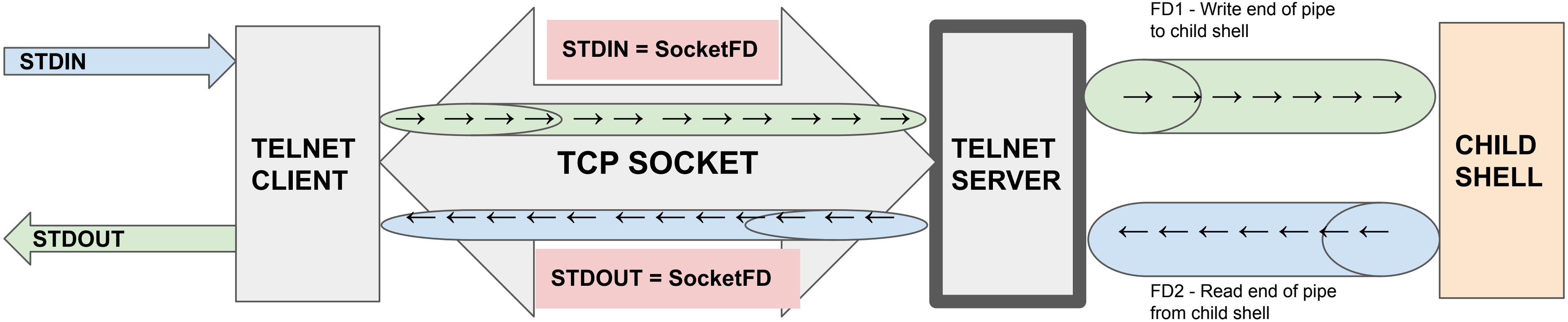
FD2 - Read end of pipe from child shell

CHILD SHELL

TCP Socket -

- Socket is again a file descriptor
- Full duplex - you can read and write from the same file descriptor

🟥 Modified file descriptors



STDIN

STDOUT

TELNET CLIENT

STDIN = SocketFD

TCP SOCKET

STDOUT = SocketFD

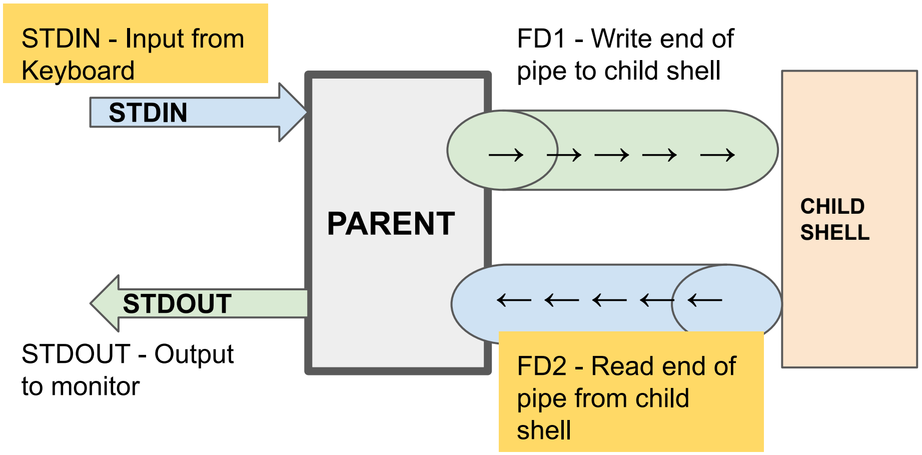
TELNET SERVER

FD1 - Write end of pipe to child shell

FD2 - Read end of pipe from child shell

CHILD SHELL

Project 1b - Poll (TELNET client)



STDIN - Input from Keyboard

STDIN

PARENT

STDOUT

STDOUT - Output to monitor

FD1 - Write end of pipe to child shell

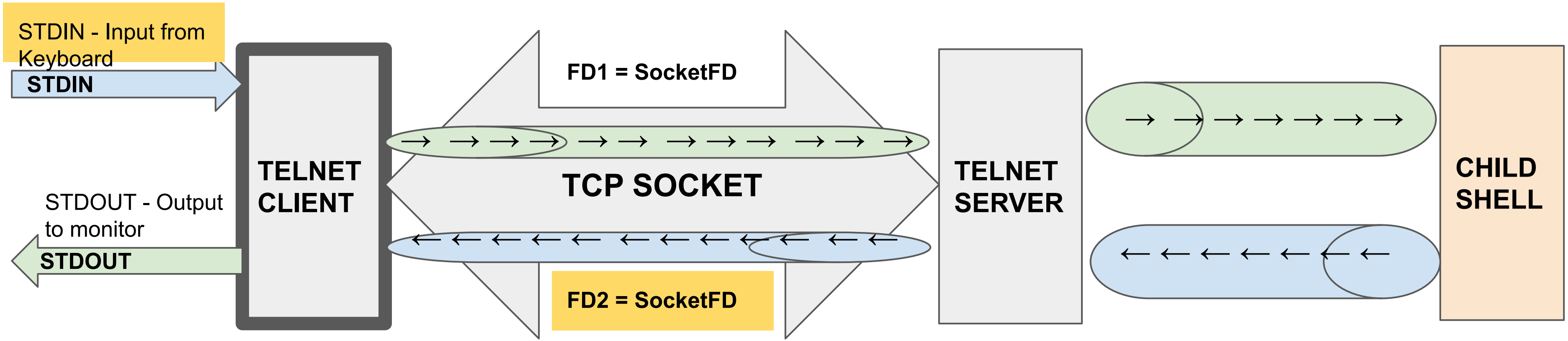
CHILD SHELL

FD2 - Read end of pipe from child shell

TCP Socket -

- Socket is again a file descriptor
- Full duplex - you can read and write from the same file descriptor

🟨 FD’s being polled



STDIN - Input from Keyboard

STDIN

TELNET CLIENT

STDOUT - Output to monitor

STDOUT

FD1 = SocketFD

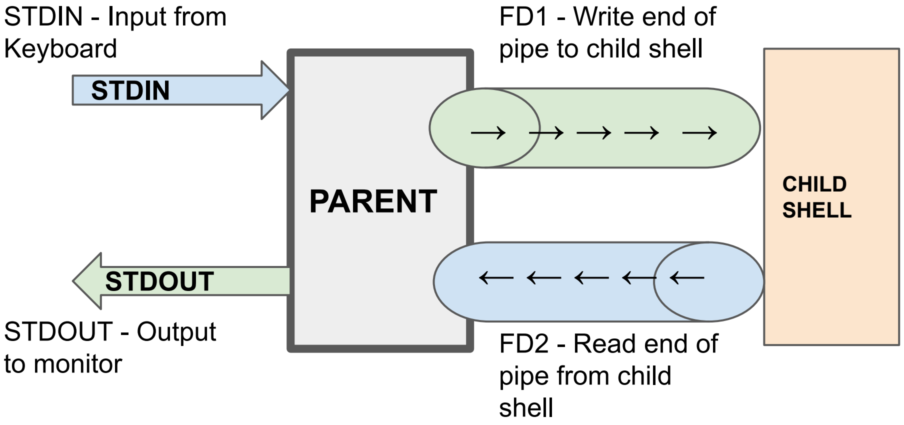
TCP SOCKET

FD2 = SocketFD

TELNET SERVER

CHILD SHELL

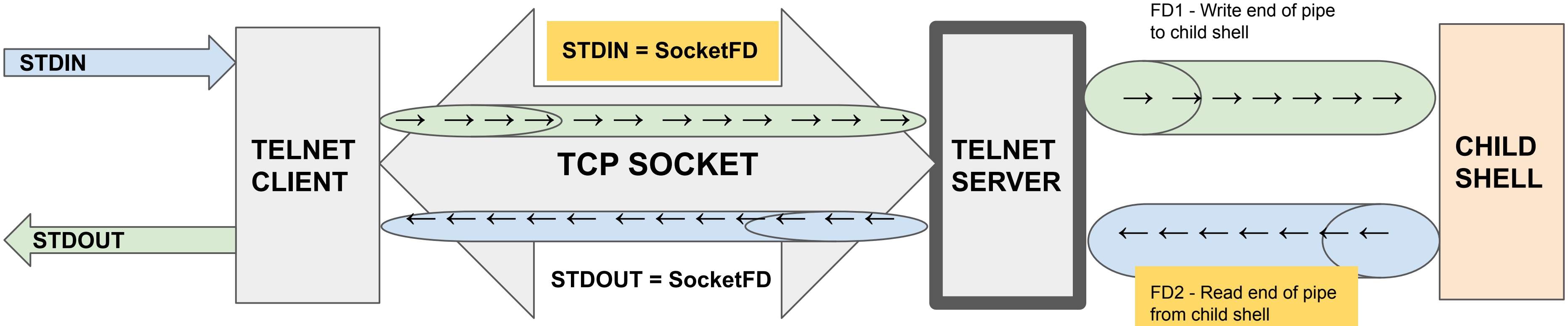
Project 1b - Poll (TELNET server)


STDIN - Input from Keyboard
STDIN
FD1 - Write end of pipe to child shell
PARENT
CHILD SHELL
STDOUT
STDOUT - Output to monitor
FD2 - Read end of pipe from child shell

TCP Socket -

- Socket is again a file descriptor
- Full duplex - you can read and write from the same file descriptor

🟨 FD's being polled



Socket Programming - Server

- Create a socket with the socket() system call
- Bind the socket to an address using the bind() system call. For a server socket on the Internet, an address consists of a port number on the host machine.
- Listen for connections with the listen() system call
- Accept a connection with the accept() system call. This call typically blocks until a client connects with the server.
- Send and receive data

Socket Programming - Client

1. Create a socket with the socket() system call
2. Connect the socket to the address of the server using the connect() system call
3. Send and receive data. There are a number of ways to do this, but the simplest is to use the read() and write() system calls.

Socket Programming Demo

Create Server.c client.c

Compile server : gcc -o server server.c

Compile client : gcc -o client client.c

Run server : ./server 8080

Run client : ./client localhost 8080

Explain - Sockets Tutorial

How will we use socket programming

Since telnet server and client need to communicate over internet:

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

We will create TCP sockets to communicate between two processes over the internet.

TCP: A protocol that ensures that you as an application programmer need not worry about network inconsistencies.

- i.e., if a network packet is dropped due to any specific reason, TCP will ensure that it will resend that packet so that client and server need not worry about unreliable networks

Follow up: Go through detailed explanation on socket programming here

The sockfd file descriptor that we mentioned in the slides, needs to be placed in accurate places in your client and server programs.

Data Compression

The purpose of compression is to reduce the amount of space data takes up.

In communications scenarios, a compressed version of data will use less bandwidth to transmit the same data than its uncompressed form would use.

On the other hand, compression will require processing at the sending and receiving end before the data can be effectively used at the receiving process.

Sometimes the benefits in reduced bandwidth are worth the costs of extra processing and sometimes they are not, so compression is not used by default in network communications.

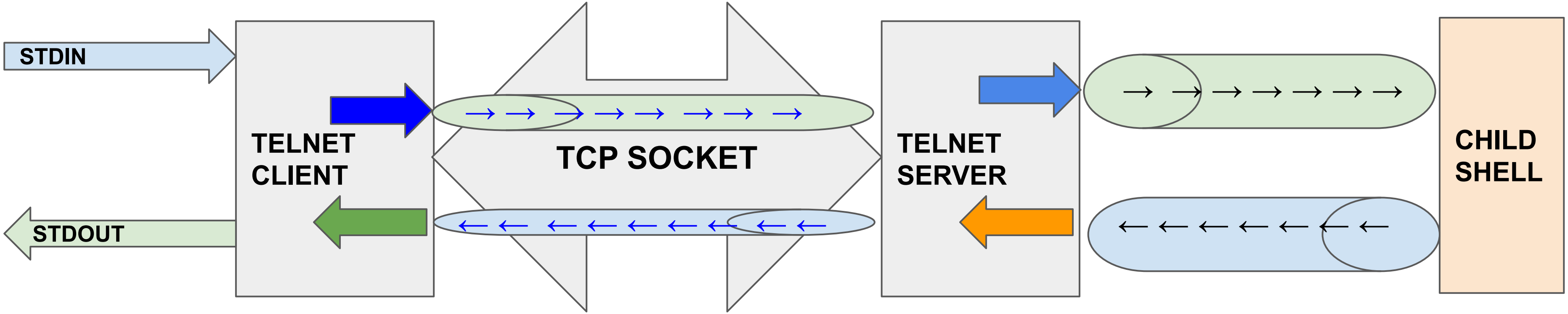
Data Compression - Server and Client

Add a --compress command line option to your client and server which, if included, will enable compression (of all traffic in both directions).

Modify both the client and server applications to compress traffic before sending it over the network and decompress it after receiving it.

Data Compression (Server and Client)

→ →	Compressed data flowing through TCP sockets
➡	Compress STDIN and send to socket
➡	Decompress socket data and send to child shell’s STDIN
⬅	Decompress socket data and send to client’s STDOUT
⬅	Compress child shell’s STDOUT and send to socket



Zlib usage

Create zpipe.c

Compile with -lz flag for including the zlib header file: `gcc -o zpipe -lz zpipe.c`

Compress a file(Deflate): `./zpipe < src > srcCompressed`

Decompress a file(Inflate): `./zpipe -d < srcCompressed > srcDecompressed`

Diff decompressed file and original file: `diff src srcDecompressed`

Zlib compression - man

Zlib Metadata: The *zlib* format has a very small header of only two bytes to identify it as a *zlib* stream and to provide decoding information, and a four-byte trailer with a fast check value to verify the integrity of the uncompressed data after decoding.

Zlib allows you to define compression levels in the range of -1 to 9. Lower compression levels result in faster execution, but less compression. Higher levels result in greater compression, but slower execution. The *zlib* constant Z_DEFAULT_COMPRESSION, equal to -1, provides a good compromise between compression and speed and is equivalent to level 6.

Zlib state is inside a struct called as z_stream_s, we are mainly responsible for these 3 fields:

- **alloc_func zalloc; /* used to allocate the internal state */**
- **free_func zfree;   /* used to free the internal state */**
- **voidpf opaque; /* private data object passed to zalloc and zfree */**

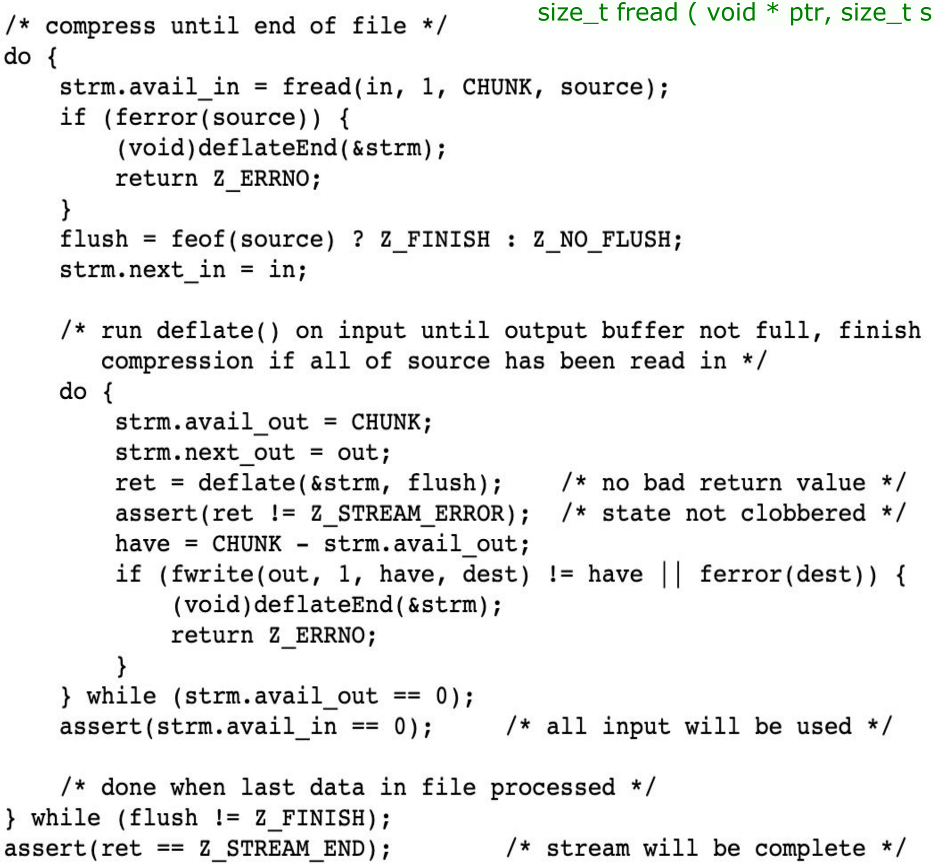
The application must initialize zalloc, zfree and opaque before calling the init function. All other fields are set by the compression library and must not be updated by the application.

When zalloc and zfree are Z_NULL on entry to the initialization function, they are set to internal routines that use the standard library functions malloc() and free().

We will set zalloc and zfree to Z_NULL

Deflate (Compress)

size_t fread (void * ptr, size_t size, size_t count, FILE * stream);



```
/* compress until end of file */
do {
    strm.avail_in = fread(in, 1, CHUNK, source);
    if (ferror(source)) {
        (void)deflateEnd(&strm);
        return Z_ERRNO;
    }
    flush = feof(source) ? Z_FINISH : Z_NO_FLUSH;
    strm.next_in = in;

    /* run deflate() on input until output buffer not full, finish
       compression if all of source has been read in */
    do {
        strm.avail_out = CHUNK;
        strm.next_out = out;
        ret = deflate(&strm, flush);    /* no bad return value */
        assert(ret != Z_STREAM_ERROR);  /* state not clobbered */
        have = CHUNK - strm.avail_out;
        if (fwrite(out, 1, have, dest) != have || ferror(dest)) {
            (void)deflateEnd(&strm);
            return Z_ERRNO;
        }
    } while (strm.avail_out == 0);
    assert(strm.avail_in == 0);     /* all input will be used */

    /* done when last data in file processed */
} while (flush != Z_FINISH);
assert(ret == Z_STREAM_END);        /* stream will be complete */
```

deflate(): takes as many of the avail_in bytes at next_in as it can process, and writes as many as avail_out bytes to next_out.

```
z_const Bytef *next_in;     /* next input byte */
uInt     avail_in;  /* number of bytes available at next_in */
uLong    total_in;  /* total number of input bytes read so far */
```

Those counters and pointers are then updated past the input data consumed and the output data written.

It is the amount of output space available that may limit how much input is consumed.

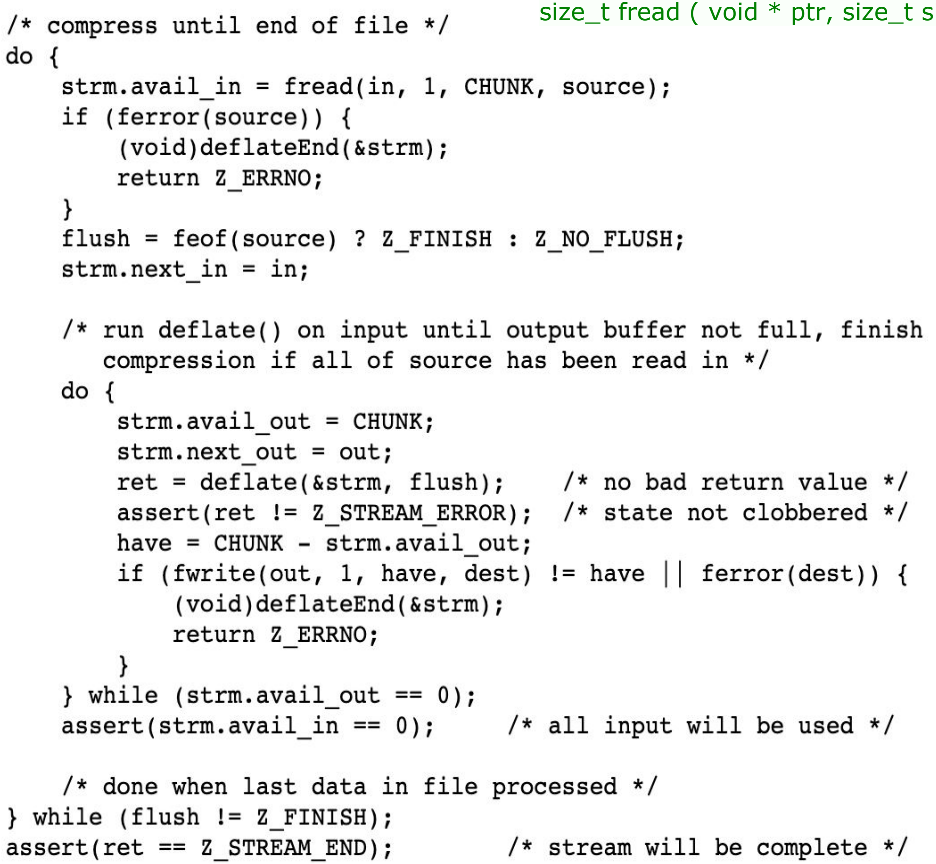
Hence the inner loop to make sure that all of the input is consumed by providing more output space each time.

```
Bytef    *next_out; /* next output byte will go here */
uInt     avail_out; /* remaining free space at next_out */
uLong    total_out; /* total number of bytes output so far */
```

Since avail_in and next_in are updated by deflate(), we don't have to mess with those between deflate() calls until it's all used up.

Deflate (Compress)

size_t fread (void * ptr, size_t size, size_t count, FILE * stream);



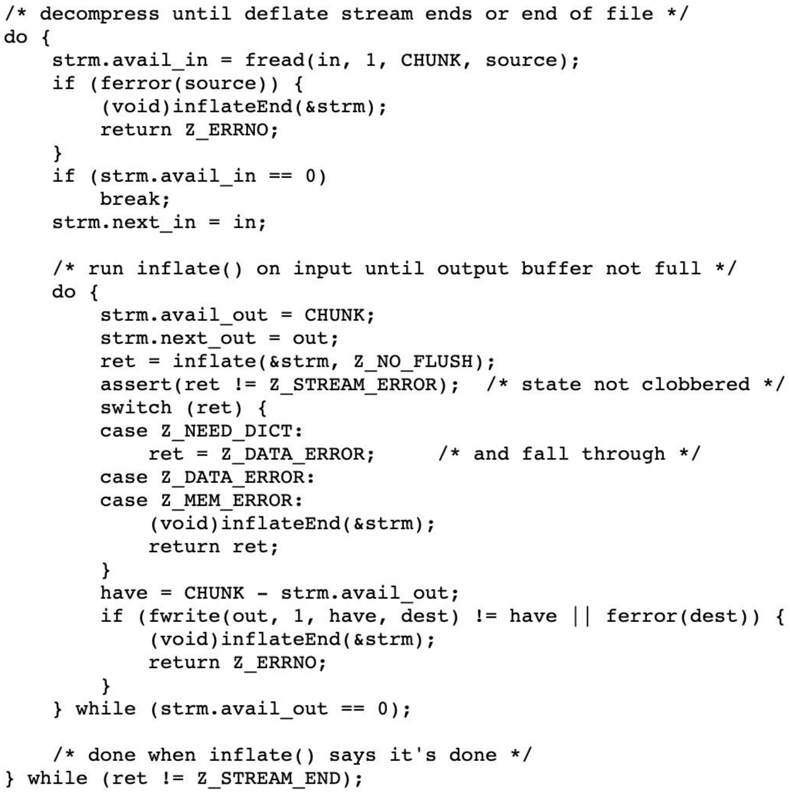
```
/* compress until end of file */
do {
    strm.avail_in = fread(in, 1, CHUNK, source);
    if (ferror(source)) {
        (void)deflateEnd(&strm);
        return Z_ERRNO;
    }
    flush = feof(source) ? Z_FINISH : Z_NO_FLUSH;
    strm.next_in = in;

    /* run deflate() on input until output buffer not full, finish
       compression if all of source has been read in */
    do {
        strm.avail_out = CHUNK;
        strm.next_out = out;
        ret = deflate(&strm, flush);    /* no bad return value */
        assert(ret != Z_STREAM_ERROR);  /* state not clobbered */
        have = CHUNK - strm.avail_out;
        if (fwrite(out, 1, have, dest) != have || ferror(dest)) {
            (void)deflateEnd(&strm);
            return Z_ERRNO;
        }
    } while (strm.avail_out == 0);
    assert(strm.avail_in == 0);     /* all input will be used */

    /* done when last data in file processed */
} while (flush != Z_FINISH);
assert(ret == Z_STREAM_END);        /* stream will be complete */
```

Dry run:

Inflate (Decompress)



```
/* decompress until deflate stream ends or end of file */
do {
    strm.avail_in = fread(in, 1, CHUNK, source);
    if (ferror(source)) {
        (void)inflateEnd(&strm);
        return Z_ERRNO;
    }
    if (strm.avail_in == 0)
        break;
    strm.next_in = in;

    /* run inflate() on input until output buffer not full */
    do {
        strm.avail_out = CHUNK;
        strm.next_out = out;
        ret = inflate(&strm, Z_NO_FLUSH);
        assert(ret != Z_STREAM_ERROR);  /* state not clobbered */
        switch (ret) {
        case Z_NEED_DICT:
            ret = Z_DATA_ERROR;     /* and fall through */
        case Z_DATA_ERROR:
        case Z_MEM_ERROR:
            (void)inflateEnd(&strm);
            return ret;
        }
        have = CHUNK - strm.avail_out;
        if (fwrite(out, 1, have, dest) != have || ferror(dest)) {
            (void)inflateEnd(&strm);
            return Z_ERRNO;
        }
    } while (strm.avail_out == 0);

    /* done when inflate() says it's done */
} while (ret != Z_STREAM_END);
```

The outer do-loop decompresses input until inflate() indicates that it has reached the end of the compressed data and has produced all of the uncompressed output.

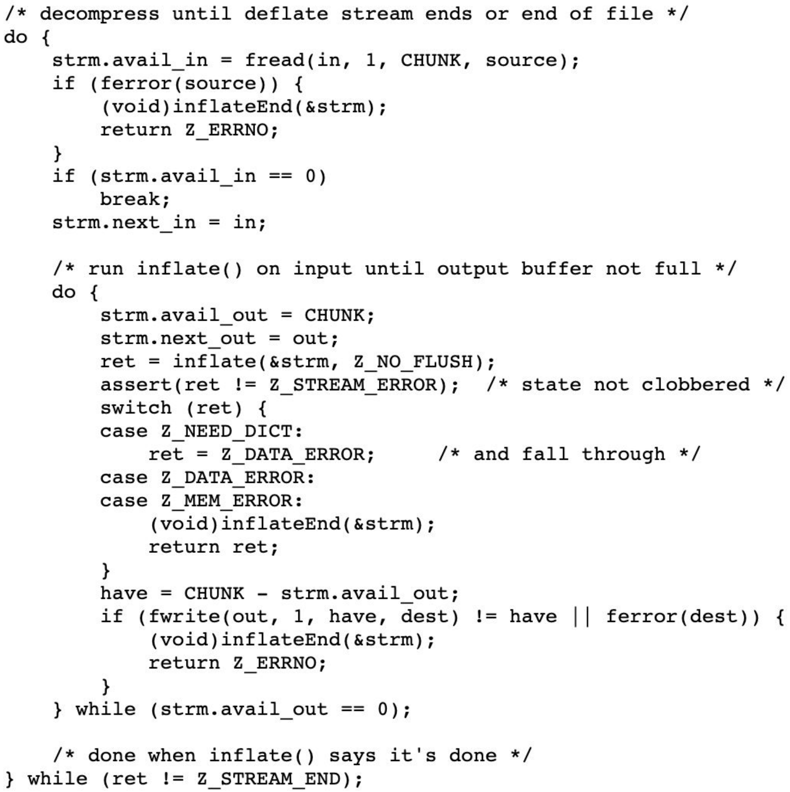
The inner do-loop has the same function it did in def(), which is to keep calling inflate() until has generated all of the output it can with the provided input.

The inner do-loop ends when inflate() has no more output as indicated by not filling the output buffer, just as for deflate(). In this case, we cannot assert that strm.avail_in will be zero, since the deflate stream may end before the file does.

The outer do-loop ends when inflate() reports that it has reached the end of the input *zlib* stream, has completed the decompression and integrity check, and has provided all of the output. This is indicated by the inflate() return value Z_STREAM_END.

The inner loop is guaranteed to leave ret equal to Z_STREAM_END if the last chunk of the input file read contained the end of the *zlib* stream. So if the return value is not Z_STREAM_END, the loop continues to read more input.

Inflate (Decompress)



```
/* decompress until deflate stream ends or end of file */
do {
    strm.avail_in = fread(in, 1, CHUNK, source);
    if (ferror(source)) {
        (void)inflateEnd(&strm);
        return Z_ERRNO;
    }
    if (strm.avail_in == 0)
        break;
    strm.next_in = in;

    /* run inflate() on input until output buffer not full */
    do {
        strm.avail_out = CHUNK;
        strm.next_out = out;
        ret = inflate(&strm, Z_NO_FLUSH);
        assert(ret != Z_STREAM_ERROR);  /* state not clobbered */
        switch (ret) {
        case Z_NEED_DICT:
            ret = Z_DATA_ERROR;     /* and fall through */
        case Z_DATA_ERROR:
        case Z_MEM_ERROR:
            (void)inflateEnd(&strm);
            return ret;
        }
        have = CHUNK - strm.avail_out;
        if (fwrite(out, 1, have, dest) != have || ferror(dest)) {
            (void)inflateEnd(&strm);
            return Z_ERRNO;
        }
    } while (strm.avail_out == 0);

    /* done when inflate() says it's done */
} while (ret != Z_STREAM_END);
```

Dry run:

Shut down order

If the client initiates the closing of the socket, it may not receive the last output from the shell. To ensure that no output is lost, shut-downs should be initiated from the server side:

1. an *exit(1)* command is sent to the shell, or the server closes the write pipe to the shell.
2. the shell exits, causing the server to receive an EOF on the read pipe from the shell.
3. the server collects and reports the shell's termination status.
4. the server closes the network socket to the client, and exits.
5. the client continues to process output from the server until it receives an error on the network socket from the server.
6. the client restores terminal modes and exits.

After reporting the shell's termination status and closing the network socket, the server should exit. Otherwise it would tie up the socket and prevent testing new server versions. After your test is complete, the client, server, and shell should all be gone. There should be no remaining orphan processes.

Server should start the shut down process after it receives a Ctrl+D

When the server receives a Ctrl+C it should send a SIGINT to shell (like in Project 1a)

The --log option

To ensure that compression is being correctly done, we will ask you to add a **--log=***filename* option to your client. If this option is specified, all data written to or read from the server should be logged to the specified file.

Prefix each log entry with **SENT # bytes:** or **RECEIVED # bytes:** as appropriate.

(Note the colon and space between the word **bytes** and the start of the data). Each of these lines should be followed by a newline character (even if the last character of the reported string was a newline).

Before running your program in a mode that creates a log file, please use the *ulimit* command to ensure that your log file does not get too large, which could happen if you have a bug in your program. *ulimit 10000* should be sufficient, but check out the ulimit man page for further details. Failing to limit the size of log files has filled up the HSSEAS Linux servers' /tmp directories in the past, which makes the machines hard to use for everybody, including you.

Sample log format output:

```
SENT 35 bytes: sendingsendingsendingsendingsending
RECEIVED 18 bytes: receivingreceiving
```

Questions?

Do you need to save and restore both clients and servers terminal modes?

Who exits first? Client or Server?

What all errors should you handle?

Polling on server and client should happen on which file descriptors?

What events should you poll on?

What flush flag is ideal for our use case?

Suggested workflow

Come up with independently working functions that

- Read and write from a socket
- Do compression and decompression from files or pipes
- Poll on specified file descriptors for given events (reuse from 1a)

Come up with a skeleton structure of the program

Fill in the blanks of the skeleton structure using the functions created in step 1.

Handle special cases (Ctrl+C, Ctrl+D, log)